UNIT - IV

FUNCTIONS

Syllabus:

Designing structured programs,
Functions – basics,
user defined functions,
Inter functions communication,
Standard functions,
Recursion-Recursive functions,
Preprocessor commands.
String –concepts,
C Strings,
String Input / Output functions,
arrays of strings,
string manipulation functions.

Source:
Bug-free Blocks (programmedlessons.org)

1. Introduction to structured programming

Software engineering is a discipline that is concerned with the construction of robust and reliable computer programs. Just as civil engineers use tried and tested methods for the construction of buildings, software engineers use accepted methods for analyzing a problem to be solved, a blueprint or plan for the design of the solution and a construction method that minimizes the risk of error.For building reliable programs, *Structured Programming* is vital. Structured programming has full computing power and greatly enhances program reliability and programmer productivity. The most productive programmers are ten times more productive than the least productive. These highly productive programmers are structured programming aces.

Bug-free Blocks:

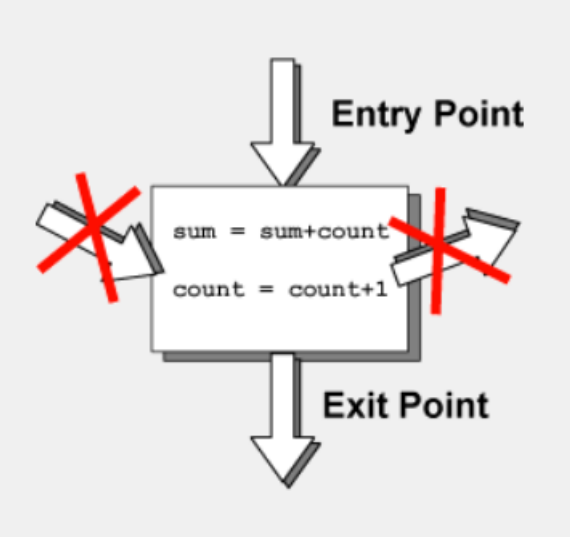
Bugs happen when control jumps out of (or into) a block of code. The block can no longer be regarded as a module that performs a particular function, but must be examined statement by statement.

Some (mostly ancient) programming languages let you jump directly into the middle of a block of code using the notorious GOTO statement. This is particularly deadly and strongly discouraged in modern programming. In assembly language it is easily possible to send control directly to nearly any point in a program. Writing assembly programs that work takes careful discipline.

You would like to think of a code block as a "black box" that performs some particular operation. Just before control enters the block everything is ready for the operation, and when control leaves the block the operation has been done correctly. In our fragment, you would like the code block to add count to sum and to increment count.

This is a prime idea of **structured programming**. A block of code is a module that has one **entry point** (where execution enters the block) and one **exit point** (where execution leaves the block). The entry point has well defined *entry conditions*. The exit point has well defined *exit conditions*.

For the block to execute correctly, execution must start at the entry point, and the entry conditions must be met. When execution leaves the block, the exit conditions are true (if the block itself is bug free).



Structured Programming Rule1: Code blocks

What if there is a bug?

If the entry conditions are correct, but the exit conditions are wrong, the bug must be in the block. This is not true if execution is allowed to jump into or out of a block. The bug might be anywhere in the program. Debugging under these conditions is much harder.

Rule 1 of Structured Programming: A code block is structured. In flow charting terms, a box with a single entry point and single exit point is structured.

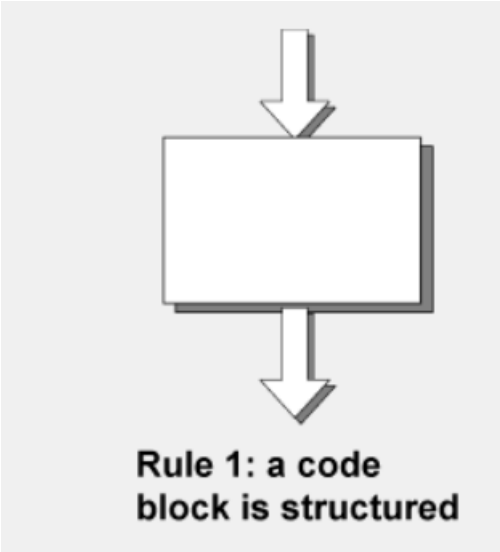
This may look obvious, but that is the idea. Structured programming is a way of making it obvious that program is correct. (Or, making it obvious where there is a problem.)

Structured programming languages show code blocks with pairs of braces, { ... }, or begin end pairs or other means.

Assembly language and older programming languages, such as early versions of FORTRAN and BASIC, were not designed with structured programming in mind. They do not have syntax for showing program blocks or the other structures of structured programming.

However, it is still possible (and essential) to do structured programming in these languages. You (the programmer) must design the code using structuring principles. Then, when coding you must follow your design.

These old languages included the notorious GOTO statement which sent control from one point in a program to any other point in a program, directly violating Rule One. This was disastrous.



Blocks as Black boxes:

A block should have one set of entry conditions and one set of exit conditions. The block should be a module that can be used for building larger structures. In C, blocks are made by putting braces around a group of statements. The code inside a block often has its own internal logic, but once the block is written and debugged it should be a black box.

```
/* Compute the square root of the discriminant of a quadratic. */
/* On Entry: a, b, and c are the coefficients of an equation that has a real-number solution. */
/* On Exit:  disc is the square root of the discriminant */

disc = b*b - 4*a*c ;
disc = sqrt( disc );
```

Each block can be thought of as a module that is part of the solution to a problem.

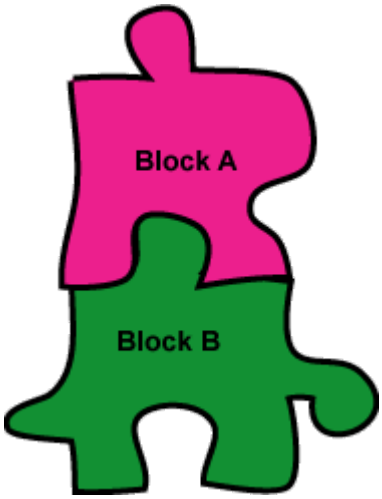
With the above two blocks, the exit conditions of the first block match the entry conditions to the second block.

Sequence:

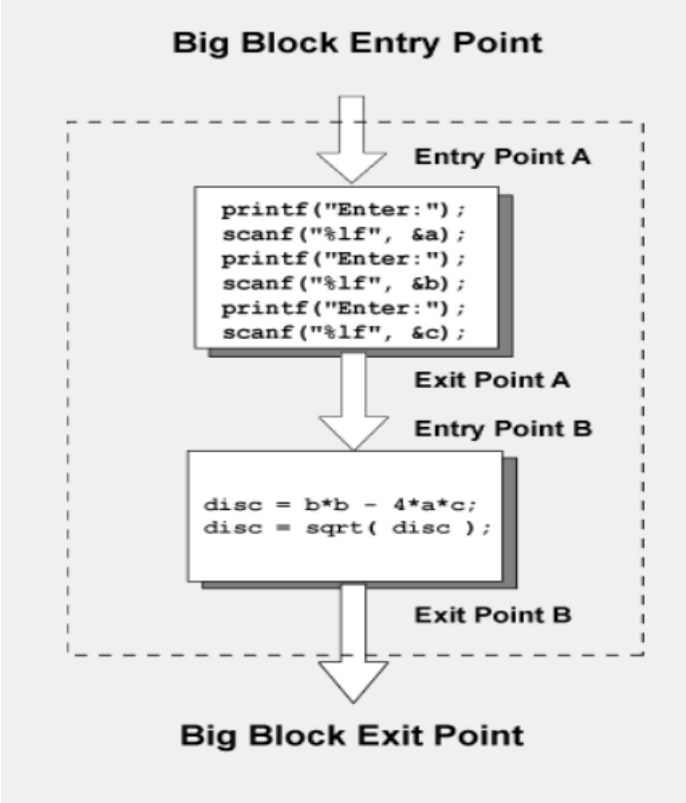
If the exit conditions of block A are the correct entry conditions for block B, then the two blocks can follow in sequence.

The set of two blocks can be regarded as one big block. If the entry conditions for block A are correct, then the exit conditions for block A are correct, then the entry conditions for block B are correct, then (finally) the exit conditions for block B are correct.

In terms of the big block, if the entry conditions for the big block are correct, then the exit conditions for the big block are also correct.



The blocks should work like jigsaw puzzle pieces where the shape of one puzzle exactly matches the shape of the following piece. Once the two pieces are fit together they can be regarded as a bigger piece that fits somewhere in the puzzle.



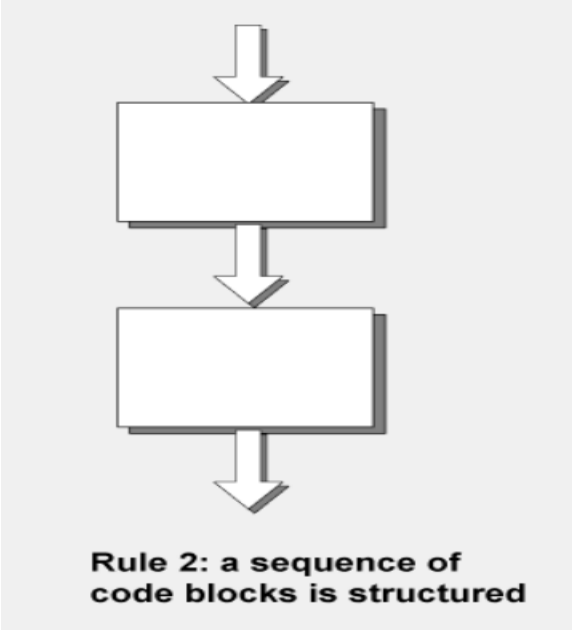
Structured Programming Rule2: Sequence

A sequence of blocks is correct if the exit conditions of each block matches the entry conditions of the following block. Control enters each block at the block's entry point, and leaves through the block's exit point. The whole sequence can be regarded as a single block, with an entry point and an exit point.

Rule 2 of Structured Programming: Two or more code blocks in sequence are structured.

Ultimately, if the program is a sequence of small blocks that fit together properly, the entire program could be thought of as one big block. When the program's entry conditions are met, then the program's exit conditions will be met.

With a program, the entry conditions consist of the problem the program was written to solve, and the exit conditions are the solution to the problem. If each of the blocks of the program are correct, then the entire program is correct.



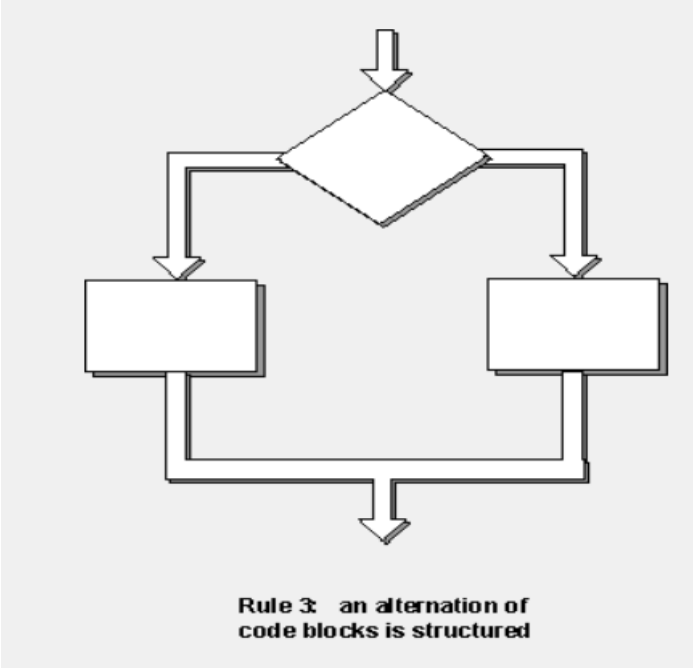
Rule 2: a sequence of code blocks is structured

Structured Programming Rule3:Alternation

If-then-else is sometimes called **alternation** (because there are alternative choices). In structured programming, each choice is a code block. If alternation is arranged as in the flowchart at right, then there is one entry point (at the top) and one exit point (at the bottom). The structure should be coded so that if the entry conditions are satisfied, then the exit conditions are fulfilled (just like a code block).

Rule 3 of Structured Programming: The alternation of two code blocks is structured.

An example of an entry condition for an alternation structure is: *the discriminant contains a value that is zero or positive*. The exit condition might be: *the correct number of roots have been computed.*. The branching structure is used to fulfil the exit condition.



Rule 3: an alternation of code blocks is structured

Implementing alternation:

In C and in most languages, alternation is implemented with an if-statement:

```
if ( condition )
  true branch
else
  false branch
```

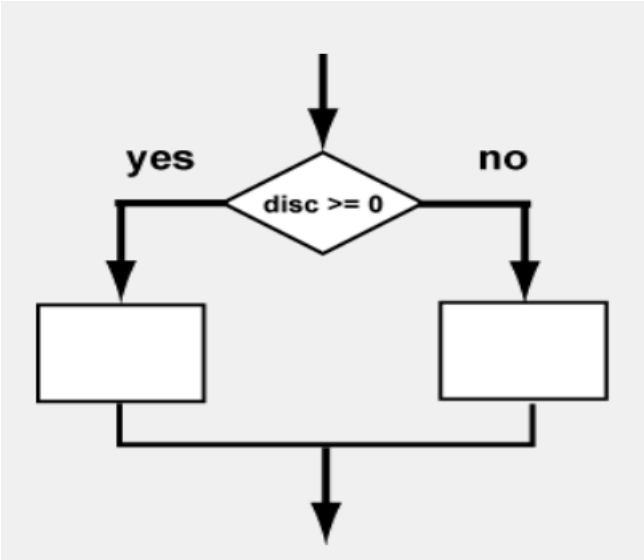
With C, the condition evaluates to 0 or to non-zero. Zero is regarded as *false* and anything non-zero is regarded as *true*. Often, the branches are code blocks:

```
if ( condition )
{
  true block
}
else
{
  false block
}
```

Here is a small example:

```
/* disc contains the discriminant of a quadratic equation */
if ( disc >= 0.0 )
{
```

```
 compute real roots
}
else
{
 compute imaginary roots
}
```



Structured Programming Rule 4: Iteration

Iteration (*while-loop*) is arranged as at right. It also has one entry point and one exit point. The entry point has conditions that must be satisfied and the exit point has conditions that will be fulfilled. There are no jumps into or out of the structure.

Rule 4 of Structured Programming: The iteration of a code block is structured.

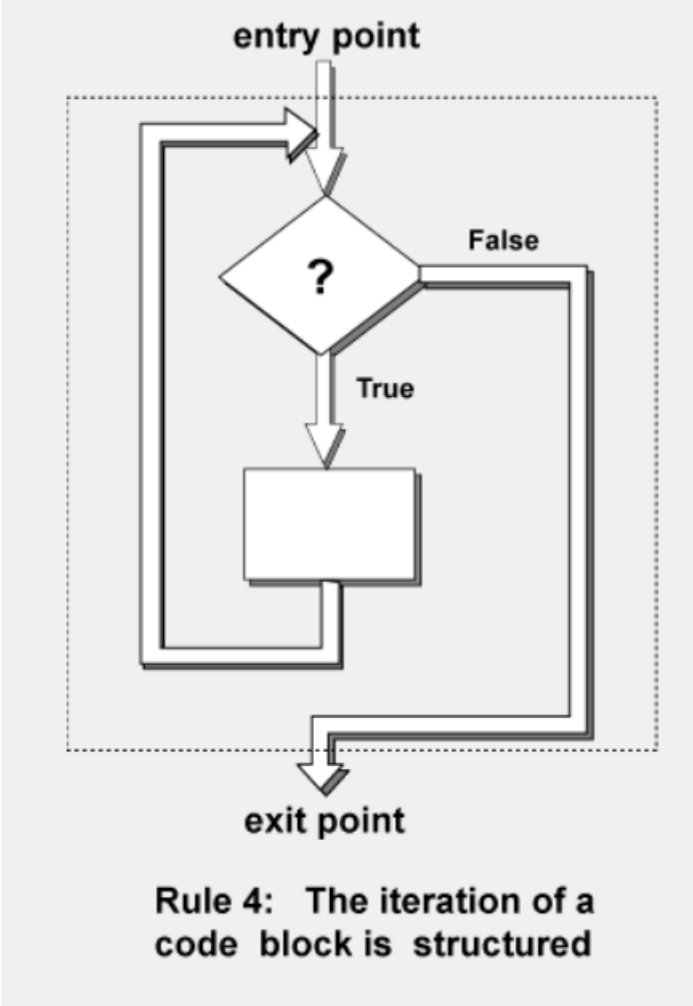
In C, iteration may be implemented with a while statement.

```
while ( condition )
 statement
```

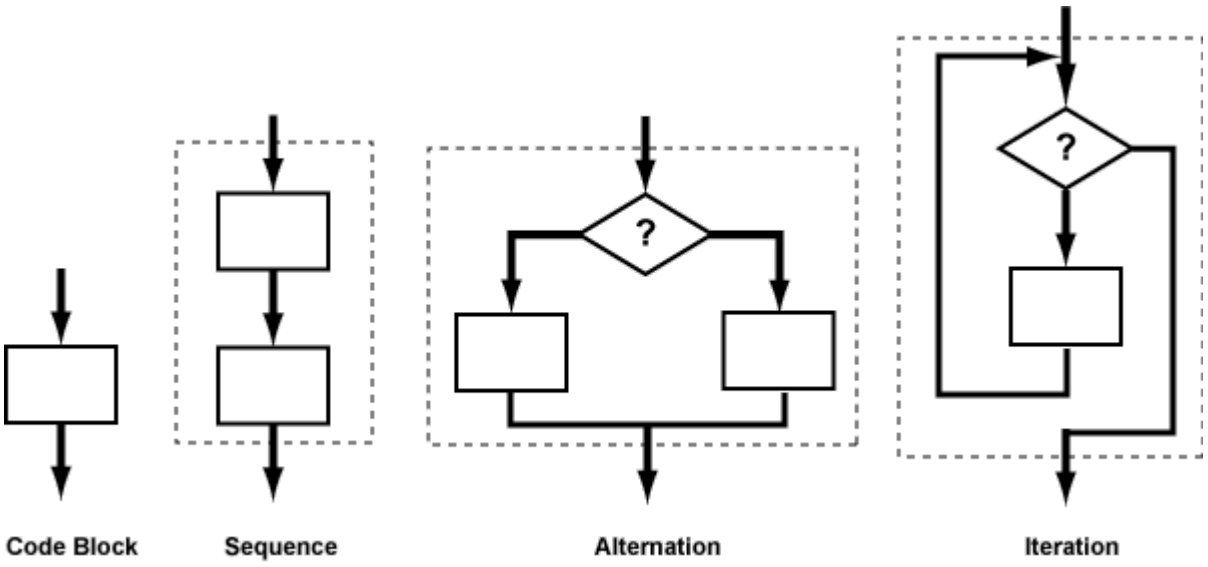
condition is an expression that evaluates to zero (regarded as *false*) or non-zero (regarded as *true*). *statement* is executed when *condition* is *true*, then control returns to the while. Almost always, the *statement* is a code block:

```
while ( condition )
{
 block
}
```

The statement or block that follows the while should ensure that *condition* will eventually become zero (false) and end the loop.



In all of the structures so far there has been an entry point at the top and an exit point at the bottom. There are no jumps into or out of the structure. The entry point has entry conditions that are expected to be satisfied, and exit conditions that are fulfilled if they are. Each of the structures can be considered to be a code block.



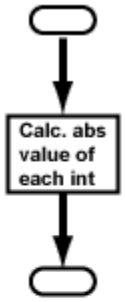
It is OK that there are several paths within a structure, but from the outside, each structure looks like a code block with one entry and one exit.

Structured Programming Rule 5: Nesting Structures

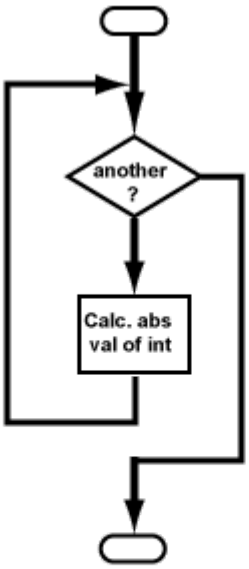
In flowcharting terms, any code block can be replaced by any of the three structures sequence, alternation, or iteration. Or, going the other direction, if there is a portion of the flowchart that has a single entry point and a single exit point, it can be summarized as a single code block.

Rule 5 of Structured Programming: A structure (of any size) that has a single entry point and a single exit point is equivalent to a code block.

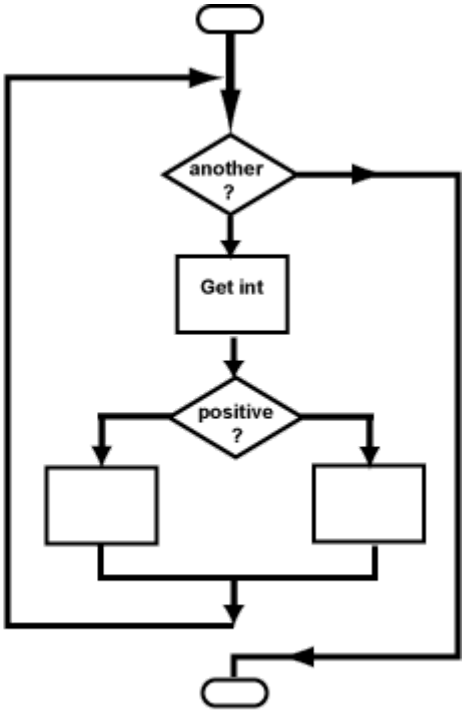
For example, say that you are designing a program to go through a list of signed integers calculating the absolute value of each one. You might (1) first regard the program as one block, then (2) sketch in the iteration required, and finally (3) put in the details of the loop body.



1. First View



2. Second View



3. Third View

Or, you might go the other way. Once the absolute value code is working, you can regard it as a single code block to be used as a component of a larger program.

The structured programming approach to program design was based on the following method.

i. To solve a large problem, break the problem into several pieces and work on each piece separately.
ii. To solve each piece, treat it as a new problem that can itself be broken down into smaller problems;
iii. Repeat the process with each new piece until each can be solved directly, without further decomposition.

2. Functions - Basics

In programming, a function is a segment that groups code to perform a specific task. A C program has at least one function main().Without main() function, there is technically no C program.

Types of C functions

There are two types of functions in C programming:

1. Library functions
2. User defined functions

1 Library functions

Library functions are the in-built function in C programming system.

For example:

main() - The execution of every C program starts form this main() function.

printf() – is used for displaying output in C.

scanf() – is used for taking input in C.

3 User defined functions

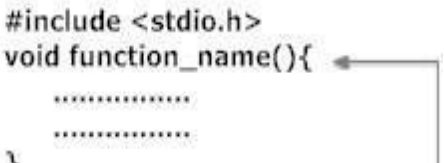
C allows programmer to define their own function according to their requirement. These types of functions are known as user- defined functions. Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he /she can create tow separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

How user-defined function works in C

Programming? #include<stdio.h>

```
voidfunction_name(){
        …….
        …….
}
int main(){
        …….
        …….
function_name();
        …….
        …….
}
```

As mentioned earlier, every C program begins form main() and program starts executing the codes inside main() function. When the control of program reaches to function_name() inside main() function. The Control of program jumps to void function_name() and executes the codes inside it. When all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function_name() from where it is called. Analyze the figure below for understanding the concept of function in C programming. Visit this page to learn in detail about user-defined functions.



Remember, the function name is an identifier and should be unique.

Advantages of user defined functions

1. User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmer working on large project can divide the workload by making different functions.

Example of user-defined function

Write a C program to add two integers. Make a function **add** to add integers and display sum int**main()** function.

```
/* program to demonstrate the working of user defined function*/
#include<stdio.h>
#include<conio.h>
int add(int a, int b);                                  /* Function Declaration */
int main()
{
        int num1,num2,sum;
        clrscr();
        printf("enter two numbers : \n ");
        scanf("%d %d",&num1,&num2);
        sum=add(num1,num2);                             /* Function
        Call*/ printf("\n sum is : %d",sum);
        getch();
}
int add(inta,int b)                                     /* Function Definition */
{
        int add;
        add=a+b;
        return add;
}
```

3.1 Function Declaration

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax:

return_typefunction_name(type(1) argument(1),. ,type(n) argument(n));

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:name of the function is add()

1. return type of the function is int.

2. two arguments of type int are passed to function.
   Function prototype are not needed if user-definition function is written before main() function.

3.2 Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked.

Syntax:

function_name(argument(1),. argument(n));

In the above example, function call is made using statement add(num1,num2); from main(). This make the control of program jump from that statement to function definition and executes the codes inside that function.

3.3 Function Definition:

Function definition contains programming codes to perform specific task.

Syntax:

```
return_typefunction_name(type(1) argument(1),..,type(n) argument(n))
{
        //body of function
}
```

Function definition has two major components:

3.3.1 Function declarator or function header

Function declarator is the first line of function definition. When a function is called, control of the program is transferred to functiondeclarator.

Syntax

return_typefunction_name(type(1) argument(1),. ,type(n) argument(n))

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body.
In above example, intadd(inta,int b) in line 12 is a function declarator.

3.3.2 Function body

Function declarator is followed by body of function inside braces.

3.4 Passing Arguments to functions

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.
In above example two variable, num1 and num2 are passed to function during function call and

these arguments are accepted by arguments a and b in function definition.

Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument *num1* was of int type and *num2* was of float type then, argument variable *a* should be of type int and b should be of type float,i.e., type of argument during function call and

function definition should be same.
A function can be called with or without an argument.

3.5 Return Statement

Return statement is used for returning a value from function definition to calling function.

Syntax:
return

(expression);

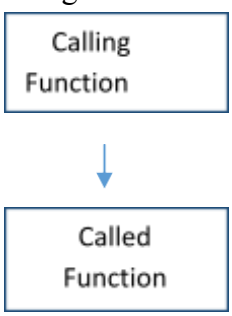
For example:

```
return a;
returna+b;
```

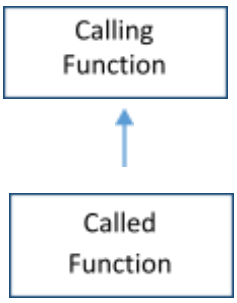
In above example, value of variable add in add() function is returned and that value is stored in variable *sum* in main() function. The data type of expression in return statement should also match the return type of function.

4. Inter Function communication

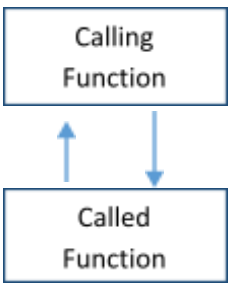
The Calling and called function are two separate entities, they need to communicate to exchange data. The data flow between the calling and called functions can be divided into three strategies



DOWNWARD FLOW



UPWARD FLOW



BI-DIRECTIONAL FLOW

Downward Flow:

In downward communication, the calling function sends data to the called function. No data flows in the opposite direction. In this strategy, copies of the data items are passed from the calling function to the called function. The called function may change the values passed, but the original values in the calling function remain untouched.

Example: **call-by-value**

```
#include<stdio.h>
```

```
        void main()
        {
                int x=2,y=3,z=0;
                add(x,y);
        }
void add(intx,int y)
        {       int z;
                z=x+y;
                printf("%d",z);
        }
```

Upward Flow:

Upward communication occurs when the called function sends data back to the called function without receiving any data from it.

Example: **call-by-reference**

```
        #include<stdio.h>
        void main()
        {
                int x=2,y=3,z=0;
                modify(&x, &y);
                printf(("%d%d",x,y);
        }
void modify(int *x,int *y)
        {
                                        *x=10;
                                        *y=20
                                }
```

Bi-directional Flow:

The strategy described for the upward direction can easily be augmented to allow the communication in both directions. The only difference is that the indirect reference must be used in both sides of the assignment variable.

```
#include<stdio.h>
void main()
{
        int x=2,y=3,z=0;
        modify(&x, &y);
        printf((“%d%d”,x,y);
}
void modify(int *x,int *y)
{
        *x=*x+10;
        *y=*y+20;
}
```

5. Standrad Library functions:

Source: *Standard Library Functions in C - Use it in Smart Way & Stand Alone in Crowd - DataFlair (data-flair.training)*

Standard Library Functions are basically the inbuilt functions in the C compiler that makes things easy for the programmer.

As we have already discussed, every C program has at least one function, that is, the **main() function**. The main() function is also a standard library function in C since it is inbuilt and conveys a specific meaning to the C compiler.

Significance:

Usability

Standard library functions allow the programmer to use the pre-existing codes available in the C compiler without the need for the user to define his own code by deriving the logic to perform certain basic functions.

Flexibility

A wide variety of programs can be made by the programmer by making slight modifications while using the standard library functions in C.

User-friendly syntax

We have already discussed in Function in C tutorial that how easy it is to grasp and use the syntax of functions.

Optimization and Reliability

All the standard library functions in C have been tested multiple times in order to generate the optimal output with maximum efficiency making it reliable to use.

Time-saving

Instead of writing numerous lines of codes, these functions help the programmer to save time by simply using the pre-existing functions.

Portability

Standard library functions are available in the C compiler irrespective of the device you are working on. These functions connote the same meaning and hence serve the same purpose regardless of the operating system or programming environment.

Library functions:

C Header Files	Description
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

<stdio.h>

This is the basic header file used in almost every program written in the C language.
It stands for standard input and standard output used to perform input-output functions, some of which are:

- **printf()**– Used to display output on the screen.
- **scanf()**– To take input from the user.
- **getchar()**– To return characters on the screen.
- **putchar()**– To display output as a single character on the screen.
- **fgets()**– To take a line as an input.
- **puts()**– To display a line as an output.
- **fopen()**– To open a file.
- **fclose()**– To close a file.

<string.h>

We have already discussed the various ***string manipulation functions in C*** in detail.

<stdlib.h>

Functions such as malloc(), calloc(), realloc() and free() can be used while dealing with dynamic memory allocation of variables.

malloc()

malloc() stands for *memory allocation*. This function is responsible for reserving a specific block of memory and returns a null pointer during the execution of the program.

calloc()

calloc stands for contiguous allocation. It is similar to malloc in all respects except the fact that it initializes the memory to 0 and has the ability to allocate numerous blocks of memory before the execution of the program.

realloc()

realloc stands for reallocation. It is used to change the size of the previously allocated memory in case the previously allocated memory is insufficient to meet the required needs of the ***variable in C***.

free()

free is responsible to free the dynamically allocated memory done by malloc(), calloc() or realloc() to the system.

<math.h>

The math header is of great significance as it is used to perform various mathematical operations such as:

- **sqrt()** – This function is used to find the square root of a number
- **pow()** – This function is used to find the power raised to that number.
- **fabs()** – This function is used to find the absolute value of a number.
- **log()** – This function is used to find the logarithm of a number.
- **sin()** – This function is used to find the sine value of a number.
- **cos()** – This function is used to find the cosine value of a number.
- **tan()** – This function is used to find the tangent value of a number.

<ctype.h>

This function is popularly used when it comes to character handling.

Some of the functions associated with <ctype.h> are:

- **isalpha()** – Used to check if the character is an alphabet or not.
- **isdigit()** – Used to check if the character is a digit or not.
- **isalnum()** – Used to check if the character is alphanumeric or not.
- **isupper()** – Used to check if the character is in uppercase or not
- **islower()** – Used to check if the character is in lowercase or not.
- **toupper()** – Used to convert the character into uppercase.
- **tolower()** – Used to convert the character into lowercase.
- **iscntrl()** – Used to check if the character is a control character or not.
- **isgraph()** – Used to check if the character is a graphic character or not.
- **isprint()** – Used to check if the character is a printable character or not
- **ispunct()** – Used to check if the character is a punctuation mark or not.
- **isspace() –** Used to check if the character is a white-space character or not.
- **isxdigit()** – Used to check if the character is hexadecimal or not.

- <ctype.h> is not supported in Linux but it works fairly well in Microsoft Windows.

<conio.h>

- It is used to perform console input and console output operations like **clrscr()** to clear the screen and **getch()** to get the character from the keyboard.
- ***Note:*** *The header file <conio.h> is not supported in Linux. It works fairly well on Microsoft Windows.*

6. Recursion

Recursion is a repetitive process in which a function calls itself. The process of repetition can be done in two ways using programming. They are

- Iterative Definition
- Recursive Definition

Iterative Definition:

Repeating a set of statements using loops is referred as Iteration.

Recursive Definition:

A repetitive function is defined recursively whenever the function appears within the definition itself. The recursive function has two elements : each call either solves one part of the problem or it reduces the size of the problem. The statement that solves the problem is known as the **base case**. The rest of the function is known as the **general case**. Each recursive function must have a base case.

Example: Finding Factorial of a number using recursive function. Let the number be 4, then

4! =4x3x2x1

If n is the number then

n!=nx(n-1)x(n-2)x(n-3)x(n-4)……x(n-n)

or

n!=nx(n-1)!

(n-1)!=(n-1) x (n-2)!

(n-2)!=(n-2)x(n-3)!

……

(n-n)!=1

4!=4x3! n=nx(n-1)! If n=4

3!=3x2! n=nx(n-1)! If n=3

2!=2x1! n=nx(n-1)! If n=2

1!=1x0! n=nx(n-1)! if n=1

0!=1 n1=1 if n=0

In the above example ,

n=nx(n-1)! Is considered as general case as it is valid for all n values

n!=1 is considered as base case as the value of 0! and 1! are 1 with which calculating the factorial terminates.

Base case: factorial(0)

General case:

nxfactorial(n-1). The execution order is as follows

4! = 4x3! (4x6=24)

3! = 3x2! (3x2=6)

2! = 2!x1 (2x1=2)

1! = 1x0! (1x1=1)

0!=1 (1=1)

The following are the rules for designing a recursive function:

1. First, determine the base case
2. Then, determine the general case
3. Finally, combine the base case and general case into a function

In combining the base and general cases into a function, we must pay careful attention to the logic. The base case, when reached, must terminate without a call to the recursive function; that is, it must execute a return.

Program:

```
#include<stdio.h>
int factorial(int );
void main()
{
        Intn,result;
        Printf("enter the
        number");
        Scanf("%d",&n);
        Result=factorial(n);
        Printf("%d",result);
}
```

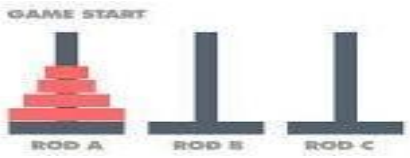
```
Intfactorial(int n)
{
        if(n==0)
                Return 1;
        else                                    Return(nxfactorial(n-1));

}
```

Limitations of Recursion:

1. Recursive solutions may involve extensive overhead because they use function calls.
2. Each time you make a call, you use up some of your memory allocation.

Example - Towers of Hanoi:




```
#include<stdio.h>
#include<conio.h>
void TOH(intn,char x, char y, char z);
void main()
{
int n;
printf("\n enter the number of plates :");
scanf("%d",&n);
TOH(n-1,"A","B","C");
getch();
}
void TOH(int n, char x, char y, char z)
{
        if(n>0)
        {                                       TOH(n-1,x,z,y);
                                                printf("\n %c -> %c",x,y);
                                                TOH(n-1,z,y,x);


        }
}
```

7. Preprocessor commands in c:

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

8. Strings - Basics

Strings are actually one-dimensional array of characters terminated by a **null**character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

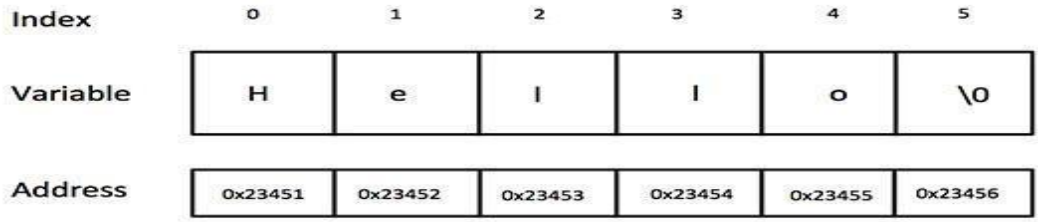
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6]={ 'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include<stdio.h>
#include<conio.h>
int main()
{
  char greeting[6]=”Hello”;
printf(“\n Greeting Message : %s \n”,greeting);
getch();
}
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

9. String input /output functions

C programming language provides many of the built-in functions to read given input and write data on screen, printer or in any file.

i. **scanf() and printf() functions**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i;
        printf(“\n Enter a value
```

```
    :"); scanf("%d",&i);
    printf("\n You entered : %d",i);
    getch();
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

NOTE : printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

```
int i = printf("studytonight");
```

In this program i will get 12 as value, because studytonight has 12 characters.

ii. getchar() &putchar() functions

The getchar()function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters. Theputchar()function prints the character passed to it on the screen and returns the same character. This function puts only single character at a time. In case you want to display more than one characters, use putchar() method in the loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int c;
        printf("Enter a character
        :"); c=getchar();
        putchar(c)
        ; getch();
}
```

When you will compile the above code,it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

iii. gets() & puts() functions

The gets() function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (end of file). The puts() function writes the string **s** and a trailing newline to stdout.

```
#include<stdio.h>
#include<conio.h
> void main()
{
        charstr[100];
        printf("Enter a String
        :"); gets(str);
        puts(str);
        getch();
}
```

When you will compile the above code,it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between scanf() and gets() functions

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.
If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it complete.

10. String handling functions:

There are any important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

1.C String Length: strlen() function

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4. char ch[20]={'R' ,'G','U','K','T','O','N','G','O', 'L','E', '\0'};
5.    printf("Length of string is: %d",strlen(ch));
6.  return 0;
7. }
```

Output:

```
Length of string is: 11
```

2.C Copy String: strcpy()

The strcpy(destination, source) function copies the source string in destination.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.  char ch[20]={'R' ,'G','U','K','T','O','N','G','O', 'L','E', '\0'};
5.    char ch2[20];
6.    strcpy(ch2,ch);
7.    printf("Value of second string is: %s",ch2);
8.  return 0;
```

```
}
```

Output:

```
Value of second string is:RGUKTONGOLE
```

3.C String Concatenation: strcat()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
   char ch2[10]={'c', '\0'};
   strcat(ch,ch2);
   printf("Value of first string is: %s",ch);
 return 0;
}
```

Output:

```
Value of first string is: helloc
```

4.C Compare String: strcmp()

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str1[20],str2[20];
   printf("Enter 1st string: ");
   gets(str1);//reads string from console
   printf("Enter 2nd string: ");
   gets(str2);
   if(strcmp(str1,str2)==0)
      printf("Strings are equal");
   else
      printf("Strings are not equal");
 return 0;
}
```

Output:

```
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal
```

5.C Reverse String: strrev()

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function

Example1: In below program, string “Hello” is reversedusing strrev( ) function and output is displayed as “olleH”

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
   printf("Enter string: ");
   gets(str);//reads string from console
   printf("String is: %s",str);
```

```
8.     printf("\nReverse String is: %s",strrev(str));
9.    return 0;
10. }
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

6.C String Lowercase: strlwr()

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
1.  #include<stdio.h>
2.  #include <string.h>
3.  int main(){
4.    char str[20];
5.    printf("Enter string: ");
6.    gets(str);//reads string from console
7.    printf("String is: %s",str);
8.    printf("\nLower String is: %s",strlwr(str));
9.   return 0;
10. }
```

Output:

```
Enter string: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint
```

7.C String Uppercase: strupr()

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

EX:1

```
1.  #include<stdio.h>
2.  #include <string.h>
3.  int main(){
4.    char str[20];
5.    printf("Enter string: ");
6.    gets(str);//reads string from console
7.    printf("String is: %s",str);
8.    printf("\nUpper String is: %s",strupr(str));
9.   return 0;
10. }
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

Array of Strings:

The string is a collection of characters, an array of a string is an array of arrays of characters. Each string is terminated with a null character. An array of a string is one of the most common applications of two-dimensional arrays.

scanf( ) is the input function with %s format specifier to read a string as input from the terminal. But the drawback is it terminates as soon as it encounters the space. To avoid this gets( ) function which can read any number of strings including white spaces.

Sting is an array of characters terminated with the special character known as the null character ("\0").

Syntax

```
datatype name_of_the_array[size_of_elements_in_array];
char str_name[size];
```

Example

```
datatype name_of_the_array [ ] = { Elements of array };
char str_name[8] = "Strings";
```

Str_name is the string name and the size defines the length of the string (number of characters).

A String can be defined as a one-dimensional array of characters, so an array of strings is two –dimensional array of characters.

Syntax

```
char str_name[size][max];
```

Syntax

```
char str_arr[2][6] = { {'g','o','u','r','i','\0'}, {'r',' a',' m','\0'}};
```

Alternatively, we can even declare it as

Syntax

```
char str_arr[2][6] ={"gouri", "ram"};
```

From the given syntax there are two subscripts first one is for how many strings to declare and the second is to define the maximum length of characters that each string can store including the null character. C concept already explains that each character takes 1 byte of data while allocating memory, the above example of syntax occupies 2 * 6 =12 bytes of memory.

Example

```
char str_name[8] = {'s','t','r','i','n','g','s','\0'};
```

By the rule of initialization of array, the above declaration can be written as

```
char str_name[] = "Strings";
```

0 1 2 3 4 5 6 7 Index

Variables 2000 2001 2002 2003 2004 2005 2006 2007 Address

This is a representation of how strings are allocated in memory for the above-declared string in C.

Each character in the string is having an index and address allocated to each character in the string. In the above representation, the null character ("\0") is automatically placed by the C compiler at the end of every string when it initializes the above-declared array. Usually, strings are declared using double quotes as per the rules of strings initialization and when the compiler encounters double quotes it automatically appends null character at the end of the string.

From the above example as we know that the name of the array points to the 0th index and address 2000 as we already know the indexing of an array starts from 0. Therefore,

```
str_name + 0 points to the character "s"
str_name + 1 points to the character "t"
```

As the above example is for one-dimensional array so the pointer points to each character of the string.

Examples of Array String in C

Following are the examples:

Example:

```
#include <stdio.h>

int main()

{

char name[10];
```

```
printf("Enter the name: ");
fgets(name, sizeof(name), stdin);
printf("Name is : ");
puts(name);
return 0;
}
```

Output:

```
String = gouri
String = ram
```

Now for two-dimensional arrays, we have the following syntax and memory allocation. For this, we can take it as row and column representation (table format).

```
char str_name[size][max];
```

In this table representation, each row (first subscript) defines as the number of strings to be stored and column (second subscript) defines the maximum length of the strings.

```
char str_arr[2][6] = { {'g','o','u','r','i','\0'}, {'r',' a',' m','\0'}};
```

Alternatively, we can even declare it as

Syntax:

```
char str_arr[2][8] ={"gouri", "ram"};
```

Index Rows	0	1	2	3	4	5	6	7
0	g	O	u	r	i	\0	\0	\0
1	r	A	m	\0	\0	\0	\0	\0

From the above example as we know that the name of the array points to the 0th string. Therefore,

str_name + 0 points to 0th string “gouri”

str_name + 1 points to 1st string “ram”

As the above example is for two-dimensional arrays so the pointer points to each string of the array.

Call by Value and Call by Reference in C

In C, a function specifies the modes of parameter passing to it. There are two ways to specify function calls: call by value and call by reference in C. In call by value, the function parameters gets the copy of actual parameters which means changes made in function parameters did not reflect in actual parameters. In call by reference, the function parameter gets reference of actual parameter which means they point to similar storage space and changes made in function parameters will reflect in actual parameters.

Introduction

Suppose you have a file and someone wants the information present in the file. So to protect from alteration in the original file, you give a copy of your file to them and if you want the changes done by someone else in your file then you have to give them your original file. In C also, if we want the changes done by function to reflect in the original parameters also, then we pass the parameter by reference, and if we don't want the changes in the original parameter, then we pass the parameters by value. We get to know about both call by value and call by reference in c and their differences in upcoming sections.

Difference Between Call by Value and Call by Reference in C

Calling by Value	Calling by Reference
Copies the value of an object.	Pass a pointer that contains the memory address of an object that gives access to its contents.
Guarantees that changes that alter the state of the parameter will only affect the named parameter bounded by the scope of the function.	Changes that alter the state of the parameter will reflect to the contents of the passed object.
Simpler to implement and simpler to reason with.	More difficult to keep track of changing values that happens for each time a function may be called.

Call by Value in C

Calling a function by value will cause the program to copy the contents of an object passed into a function. To implement this in C, a function declaration has the following form: [return type] functionName([type][parameter name],...).

Call by Value Example: Swapping the values of the two variables

```c
#include <stdio.h>

void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 10;
```

```
    int y = 11;
    printf("Values before swap: x = %d, y = %d\n", x,y);
    swap(x,y);
    printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output:

```
Values before swap: x = 10, y = 11
Values after swap: x = 10, y = 11
```

We can observe that even when we change the content of x and y in the scope of the swap function, these changes do not reflect on x and y variables defined in the scope of main. This is because we call swap() by value and it will get separate memory for x and y so the changes made in swap() will not reflect in main().

Call by Reference in C

Calling a function by reference will give function parameter the address of original parameter due to which they will point to same memory location and any changes made in the function parameter will also reflect in original parameters. To implement this in C, a function declaration has the following form: [return type] functionName([type]* [parameter name],...).

Call by Reference Example: Swapping the values of the two variables

```
#include <stdio.h>

void swap(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int x = 10;
    int y = 11;
    printf("Values before swap: x = %d, y = %d\n", x,y);
    swap(&x,&y);
    printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output:

```
Values before swap: x = 10, y = 11
Values after swap: x = 11, y = 10
```

We can observe in function parameters instead of using int x,int y we used int *x,int *y and in function call instead of giving x,y, we give &x,&y this methodology is call by reference as we used pointers as function parameter which will get original parameters' address instead of their value. & operator is used to give address of the variables and * is used to access the memory location that pointer is pointing. As the function variable is pointing to the same memory location as the original parameters, the changes made in swap() reflect in main() which we can see in the above output.

When to Use Call by Value and Call by Reference in C?

Copying is expensive, and we have to use our resources wisely. Imagine copying a large object like an array with over a million elements only to enumerate the values inside the array, doing so will result in a waste of time and memory. Time is valuable and we can omit to copy when:

1. We intend to read state information about an object, or
2. Allow a function to modify the state of our object.

However, when we do not intend our function to alter the state of our object outside of our function, copying prevents us from making unintentional mistakes and introduce bugs. Now we know when to use call by value and call by reference in C.

Now we will discuss the advantages and disadvantages of call by value and call by reference in C.

Advantages of Using Call by Value Method

- Guarantees that changes that alter the behavior of a parameter stay within its scope and do not affect the value of an object passed into the function
- Reduce the chance of introducing subtle bugs which can be difficult to monitor.
- Passing by value removes the possible side effects of a function which makes your program easier to maintain and reason with.

Advantages of Using Call by Reference Method

- Calling a function by reference does not incur performance penalties that copying would require. Likewise, it does not duplicate the memory necessary to access the content of an object that resides in our program.
- Allows a function to update the value of an object that is passed into it.
- Allows you to pass functions as references through a technique called function pointers which may alter the behavior of a function. Likewise, lambda expressions may also be passed inside a function. Both enable function composition which has neat theoretical properties.

Disadvantages of Using Call by Value Method

- Incurs performance penalty when copying large objects.
- Requires to reallocate memory with the same size as the object passed into the function.

Disadvantages of Using Call by Reference Method

- For every function that shares with the same object, your responsibility of tracking each change also expands.
- Making sure that the object does not die out abruptly is a serious issue about calling a function by reference. This is especially true in the context of a multithreaded application.

Conclusion

- There are two ways to pass an argument in C: passing by value and passing by reference. Also known as call by value and call by reference in C.
- Passing by value copies the content of an object that is being passed into the function. This results in an independent object that exists within the scope of that function. This provides a simpler way to think and reason with our program as we do not allow the function to modify the contents of an object.
- Passing by reference elides copying and instead passes the memory address of an object. A function may be granted with the privileges to modify the values of an object that is passed onto it.